SCALABLE GRAPH ANALYTICS

ERHARD RAHM

www.scads.de
Management and Analysis of Big Graph Data: Current Systems and Open Challenges

Martin Junghanns, André Petermann, Martin Neumann and Erhard Rahm

Abstract Many big data applications in business and science require the management and analysis of huge amounts of graph data. Suitable systems to manage and to analyze such graph data should meet a number of challenging requirements including support for an expressive graph data model with heterogeneous vertices and
Facebook
ca. 1.3 billion users
c. 340 friends per user

Twitter
ca. 300 million users
ca. 500 million tweets per day

Internet
ca. 2.9 billion users

Gene (human)
20,000-25,000
ca. 4 million individuals
Patients
> 18 millions (Germany)
Illnesses
> 30,000

World Wide Web
ca. 1 billion Websites

LOD-Cloud
ca. 90 billion triples
"GRAPHS ARE EVERYWHERE"

Graph = (Vertices, Edges)
"GRAPHS ARE EVERYWHERE"

Graph = (Users, Followers)
"GRAPHS ARE EVERYWHERE"

Graph = (Users, Friendships)
"GRAPHS ARE HETEROGENEOUS"

Graph = (Users ∪ Bands, Friendships ∪ Likes)
“GRAPHS CAN BE ANALYZED”

Graph = (Users ∪ Bands, Friendships ∪ Likes)
Graphs can be analyzed

Assuming a social network
1. Determine subgraph
2. Find communities
3. Filter communities
4. Find common subgraph
GRAPH DATA ANALYTICS: HIGH-LEVEL REQUIREMENTS

- **all „V“ challenges**
  - volume (scalability)
  - Variety (support for heterogenous data / data integration)
  - Velocity (dynamically changing graph data)
  - veracity (high data quality)
  - value (improved business value)

- **ease-of-use**

- **high cost-effectiveness**
GRAPH DATA ANALYTICS: REQUIREMENTS

- powerful but easy to use graph data model
  - support for heterogeneous, schema-flexible vertices and edges
  - support for collections of graphs (not only 1 graph)
  - powerful graph operators
- powerful query and analysis capabilities
  - interactive, declarative graph queries
  - scalable graph mining
- high performance and scalability
- persistent graph storage and transaction support
- graph-based integration of many data sources
- versioning and evolution (dynamic/temporal graphs)
- comprehensive visualization support
AGENDA

- Motivation
  - graph data
  - requirements

- Graph data systems
  - graph database systems
  - distributed graph processing systems (Pregel, etc.)
  - distributed graph dataflow systems (GraphX, Gelly)

- Gradoop
  - architecture
  - Extended Property Graph Model (EPGM)
  - implementation and performance evaluation

- Open challenges
GRAPH DATABASES

- First Generation:
  - research prototypes only
  - peak popularity in early 90s

- Second Generation:
  - NoSQL movement
  - commercial systems
graph data model
- mostly property graphs, RDF or generic graphs
- different vertex and edge data
- graph operators (traversal, pattern matching) / queries

application scope
- mostly queries/OLTP on small graph portions
- some support for analytical queries/computations (analyze whole graph, e.g., page rank)
- popular data model for commercial graph DBMS
  - de-facto industry standard TinkerPop
  - query languages Gremlin (TinkerPop) and Cypher (Neo4j, openCypher)

- query example (pattern matching)

MATCH (x:User)-[a:knows]->(y:User),
  (x)-[b:memberOf]->(z:Group),
  (y)-[c:memberOf]->(z)
WHERE x.age < 25 AND z.name = 'GDM' AND
  a.since < 2016 AND c.since >= 2016
RETURN y.name, x.name

(a) Pattern graph

(b) Cypher
<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Scope</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDF/SPARQL</td>
<td>OLTP/Queries</td>
<td>Approach</td>
</tr>
<tr>
<td></td>
<td>PGM/TinkerPop</td>
<td>Analytics</td>
<td>Replication</td>
</tr>
<tr>
<td></td>
<td>Generic</td>
<td></td>
<td>Partitioning</td>
</tr>
<tr>
<td>Apache Jena TBD</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>AllegroGraph</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>MarkLogic</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>Ontotext GraphDB</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>Oracle Spatial and Graph</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Virtuoso</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TripleBit</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Blazegraph</td>
<td>✓/✓ ✓/✓</td>
<td>✓</td>
<td>native RDF</td>
</tr>
<tr>
<td>IBM System G</td>
<td>✓/✓ ✓/✓</td>
<td>✓</td>
<td>native PGM,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>wide column store</td>
</tr>
<tr>
<td>Stardog</td>
<td>✓/✓ ✓/✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SAP Active Info. Store</td>
<td>✓/-</td>
<td>✓</td>
<td>relational</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>✓/✓</td>
<td>✓</td>
<td>document store</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>Neo4j</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Oracle Big Data</td>
<td>✓/✓</td>
<td>✓</td>
<td>key value store</td>
</tr>
<tr>
<td>OrientDB</td>
<td>✓/✓</td>
<td>✓</td>
<td>document store</td>
</tr>
<tr>
<td>Sparksee</td>
<td>✓/✓</td>
<td>✓</td>
<td>native</td>
</tr>
<tr>
<td>SQLGraph</td>
<td>✓/✓</td>
<td>✓</td>
<td>relational</td>
</tr>
<tr>
<td>Titan</td>
<td>✓/✓ ✓/-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HypergraphDB</td>
<td>✓</td>
<td>✓</td>
<td>native</td>
</tr>
</tbody>
</table>

**SYSTEM COMPARISON**
## COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>Graph Database Systems Neo4j, OrientDB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data model</td>
<td>rich graph models (PGM)</td>
<td></td>
</tr>
<tr>
<td>focus</td>
<td>queries</td>
<td></td>
</tr>
<tr>
<td>query language</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>graph analytics</td>
<td>(no)</td>
<td></td>
</tr>
<tr>
<td>scalability</td>
<td>vertical</td>
<td></td>
</tr>
<tr>
<td>analysis workflows</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>persistency</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>dynamic graphs / versioning</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>data integration</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>visualization</td>
<td>(yes)</td>
<td></td>
</tr>
</tbody>
</table>
goal: better support for scalable/distributed graph mining
- page rank, connected components, clustering, frequent subgraphs, ...
- mostly generic graphs only (e.g., directed multigraphs)

early approaches based on MapReduce
- iterative processing via control program and multiple MR programs
- unintuitive programming and limited performance (high communication and I/O costs)

„newer“ computation models pioneered by Google Pregel
- vertex-centric programming („Think Like a Vertex“)
- Bulk-synchronous-parallel (BSP) computation
- In-memory storage of graph data
VERTEX-CENTRIC PROCESSING

- parallel and synchronized execution of vertex *compute* function
- vertex keeps state about itself
- compute function
  - reads incoming messages,
  - updates vertex state (value)
  - sends information to neighboring vertices
- vertices can deactivate themselves (call `voteToHalt()` function)
- iterative execution within *supersteps* until there are no active vertices or messages anymore (bulk-synchronous-parallel execution)
EXAMPLE – MAXIMUM VALUE

\[ S_0 \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \]

\( v = 3 \quad v = 6 \quad v = 2 \quad v = 1 \)

\[ S_1 \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \]

\( v = 6 \quad v = 6 \quad v = 2 \quad v = 6 \)

\[ S_2 \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \]

\( v = 6 \quad v = 6 \quad v = 6 \quad v = 6 \)

\[ S_3 \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \]

\( v = 6 \quad v = 6 \quad v = 6 \quad v = 6 \)
alternate execution models

- partition-centric ("Think-like-a-graph"): synchronized execution of compute functions for entire partitions (all vertices on one worker)
- asynchronous: to avoid many idle vertices/workers with skewed degree distributions

Gather-Apply-Scatter (GAS) programming model

- gather function: aggregates/combines messages
- apply function: preprocesses incoming messages and updates vertex state
- scatter function: uses vertex state to produce outgoing messages
- Goals: reduce network traffic and better workload balancing for graphs with highly skewed degree distribution

Scatter-Gather programming model

- user provides vertex and edge functions:
  - vertex function uses all incoming messages to modify vertex value
  - edge function uses vertex value to generate a message
- susceptible to execution skew (like vertex-centric)
## Graph Processing Systems

<table>
<thead>
<tr>
<th>Language</th>
<th>Computation Model</th>
<th>BSP</th>
<th>async.</th>
<th>Agg.</th>
<th>Add</th>
<th>Remove</th>
<th>Comb.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Giraph</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPS</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Mizan</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>GraphLab</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>GraphChi</td>
<td>C++, Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>Signal/Collect</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>Chaos</td>
<td>Java</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>Giraph++</td>
<td>Java</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphX</td>
<td>Scala, Java</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td>Gelly</td>
<td>Scala, Java</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n.a.</td>
</tr>
<tr>
<td></td>
<td><strong>Graph Database Systems</strong></td>
<td><strong>Graph Processing Systems</strong> (Pregel, Giraph)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------</td>
<td>----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data model</td>
<td>rich graph models (PGM)</td>
<td>generic graph models</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>focus</td>
<td>queries</td>
<td>analytic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>query language</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph analytics</td>
<td>no</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scalability</td>
<td>vertical</td>
<td>horizontal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workflows</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>persistency</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic graphs / versioning</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data integration</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>visualization</td>
<td>(yes)</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph processing systems are specialized systems
   - tailored programming abstractions for fast execution of a single iterative graph algorithm

Complex analytical problems often require the combination of multiple techniques, e.g.:
   - creation of combined graph structures from different sources (data extraction, transformation and integration)
   - different analysis steps: queries, iterative graph processing, machine learning, ...

Dataflow systems can combine such tasks within dataflow programs/workflows/scripts for distributed execution
   - 1st generation: MapReduce workflows
   - Apache Spark/Flink: in-memory dataflow systems
Distributed in-memory dataflow systems (e.g., Apache Spark, Apache Flink)

- general-purpose operators (e.g. map, reduce, filter, join) => 
  transformations
- specialized libraries (e.g. machine learning, graph analysis)
- holistic view enables optimizations (operator reordering, caching, etc.)

- **Dataset** := distributed collection of data objects
- **Transformation** := operation on datasets (higher-order function)
- **Dataflow Program** := composition of transformations
Graph abstraction on top of a dataflow system (e.g., Gelly on Apache Flink and GraphX on Apache Spark)
- generic graph representation
- graph operations / transformations / processing

Graph transformations / operations
- **mutation**: adding / removing of vertices and edges
- **map**: modification of vertex and edge values
- **subgraph**: find subgraph for user-defined vertex / edge predicates
- **join**: combination of vertex / edge datasets with other datasets
- **union/difference/intersect**: combine two graphs into one

Graph processing
- Gelly implements **Pregel, GAS, Scatter-Gather** by using native Flink iteration functions
- GraphX implements **GAS** based on Spark Iterations
## COMPARISON (3)

<table>
<thead>
<tr>
<th></th>
<th><strong>Graph Database Systems</strong> (Neo4j, OrientDB)</th>
<th><strong>Graph Processing Systems</strong> (Pregel, Giraph)</th>
<th><strong>Graph Dataflow Systems</strong> (Flink Gelly, Spark GraphX)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data model</strong></td>
<td>rich graph models (PGM)</td>
<td>generic graph models</td>
<td>generic graph models</td>
</tr>
<tr>
<td><strong>focus</strong></td>
<td>queries</td>
<td>analytic</td>
<td>analytic</td>
</tr>
<tr>
<td><strong>query language</strong></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>graph analytics</strong></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>scalability</strong></td>
<td>vertical</td>
<td>horizontal</td>
<td>horizontal</td>
</tr>
<tr>
<td><strong>Workflows</strong></td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>persistency</strong></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>dynamic graphs / versioning</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>data integration</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>visualization</strong></td>
<td>(yes)</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
An end-to-end framework for scalable (distributed) graph data management and analytics supporting a rich graph data model and queries
Motivation
- graph data
- requirements

Graph data systems
- graph database systems
- distributed graph processing systems (Pregel, etc.)
- distributed graph dataflow systems (GraphX, Gelly)

Gradoop
- architecture
- Extended Property Graph Model (EPGM)
- implementation and performance evaluation

Open challenges
GRADOOP CHARACTERISTICS

- Hadoop-based framework for graph data management and analysis
  - persistent graph storage in scalable distributed store (Hbase)
  - utilization of powerful dataflow system (Apache Flink) for parallel, in-memory processing

- Extended property graph data model (EPGM)
  - operators on graphs and sets of (sub) graphs
  - support for semantic graph queries and mining

- declarative specification of graph analysis workflows
  - Graph Analytical Language - GrALa

- end-to-end functionality
  - graph-based data integration, data analysis and visualization

- open-source implementation: www.gradoop.org
● Integrate data from one or more sources into a dedicated graph store with common graph data model

● Definition of analytical workflows from operator algebra

● Result representation in meaningful way
HIGH LEVEL ARCHITECTURE

Data flow
Control flow

Workflow Declaration
Visual
GrALa DSL

Representation

Extended Property Graph Model

Flink Operator Implementations
Data Integration
Graph Analytics

Representation

Flink Operator Execution

HBase Distributed Graph Store

HDFS/YARN Cluster
EXTENDED PROPERTY GRAPH MODEL (EPGM)

- includes PGM as special case
- support for collections of logical graphs / subgraphs
  - can be defined explicitly
  - can be result of graph algorithms / operators
- support for graph properties
- powerful operators on both graphs and graph collections
- Graph Analytical Language – GrALa
  - domain-specific language (DSL) for EPGM
  - flexible use of operators with application-specific UDFs
  - plugin concept for graph mining algorithms
• Vertices and directed Edges
• Vertices and directed Edges
• Logical Graphs
• Vertices and directed Edges
• Logical Graphs
• Identifiers
• Vertices and directed Edges
• Logical Graphs
• Identifiers
• Type Labels
- Vertices and directed Edges
- Logical Graphs
- Identifiers
- Type Labels
- Properties
Operators
Operators

Unary
- Aggregation
- Pattern Matching
- Transformation
- Grouping
- Subgraph
- Call *

Binary
- Combination
- Overlap
- Exclusion
- Equality
- Union
- Intersection
- Difference
- Equality

Algorithms
- Gelly Library
- BTG Extraction
- Adaptive Partitioning
- Frequent Subgraphs

Graph Collection
- Selection
- Distinct
- Sort
- Limit
- Apply *
- Reduce *
- Call *

* auxiliary
LogicalGraph graph3 = graph1.combine(graph2);
LogicalGraph graph4 = graph1.overlap(graph2);
LogicalGraph graph5 = graph1.exlude(graph2);
```python
udf = (graph => graph['vertexCount'] = graph.vertices.size())
graph3 = graph3.aggregate(udf)
```
LogicalGraph graph4 = graph3.subgraph((vertex => vertex[:label] == 'green'))
LogicalGraph graph5 = graph3.subgraph((edge => edge[:label] == 'blue'))
LogicalGraph graph6 = graph3.subgraph((vertex => vertex[:label] == 'green'), (edge => edge[:label] == 'orange'))
GraphCollection collection = graph3.match("(:Green)-[:orange]->(:Orange)");
LogicalGraph grouped = graph3.groupBy(
    [:label], // vertex keys
    [:label]) // edge keys
LogicalGraph grouped = graph3.groupBy([:label], [COUNT()], [:label], [MAX('a')])
GROUPING: TYPE LEVEL (SCHEMA GRAPH)

```chef
vertexGrKeys = [:label]
edgeGrKeys = [:label]
sumGraph = databaseGraph.groupBy(vertexGrKeys, [COUNT()], edgeGrKeys, [COUNT()])
```
GROUPING: PROPERTY-SPECIFIC

personGraph = databaseGraph.subgraph((vertex => vertex[:label] == 'Person'),
                                      (edge => edge[:label] == 'knows'))
vertexGrKeys = [:label, "city"]
edgeGrKeys = [:label]
sumGraph = personGraph.groupby(vertexGrKeys, [COUNT()], edgeGrKeys, [COUNT()])
GraphCollection filtered = collection.select((graph => graph['vertexCount'] > 4));
GraphCollection frequentPatterns = collection.callForCollection(new TransactionalFSM(0.5))
Implementation and evaluation
EPGMGraphHead

Id | Label | Properties
---|-------|-------------

EPGMVertex

Id | Label | Properties | Graphs
---|-------|------------|-----

EPGMEdge

Id | Label | Properties | SourceId | TargetId | Graphs
---|-------|------------|----------|----------|-----

GradoopId := UUID
128-bit

PropertyList := List<Property>
Property := (String, PropertyValue)
PropertyValue := byte[]

GradoopIdSet := Set<GradoopId>

---

**POJO**

**DataSet<EPGMGraphHead>**

**DataSet<EPGMVertex>**

**DataSet<EPGMEdge>**
GROUP REPRESENTATION: EXAMPLE

**DataSets<EPGMGraphHead>**

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Properties</th>
<th>Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Community</td>
<td>{interest: Heavy Metal}</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>Community</td>
<td>{interest: Hard Rock}</td>
<td>{1}</td>
</tr>
</tbody>
</table>

**DataSets<EPGMVertex>**

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Properties</th>
<th>Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Person</td>
<td>{name:Alice, born:1984}</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>Band</td>
<td>{name: Metallica, founded: 1981}</td>
<td>{1}</td>
</tr>
<tr>
<td>3</td>
<td>Person</td>
<td>{name: Bob}</td>
<td>{1,2}</td>
</tr>
<tr>
<td>4</td>
<td>Band</td>
<td>{name: AC/DC, founded: 1973}</td>
<td>{2}</td>
</tr>
<tr>
<td>5</td>
<td>Person</td>
<td>{name: Eve}</td>
<td>{2}</td>
</tr>
</tbody>
</table>

**DataSets<EPGMEdge>**

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Source</th>
<th>Target</th>
<th>Properties</th>
<th>Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>likes</td>
<td>1</td>
<td>2</td>
<td>{since:2014}</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>likes</td>
<td>3</td>
<td>2</td>
<td>{since:2013}</td>
<td>{1}</td>
</tr>
<tr>
<td>3</td>
<td>likes</td>
<td>3</td>
<td>4</td>
<td>{since:2015}</td>
<td>{2}</td>
</tr>
<tr>
<td>4</td>
<td>knows</td>
<td>3</td>
<td>5</td>
<td>{}</td>
<td>{2}</td>
</tr>
<tr>
<td>5</td>
<td>likes</td>
<td>5</td>
<td>4</td>
<td>{since:2014}</td>
<td>{2}</td>
</tr>
</tbody>
</table>
// input: firstGraph (G[1]), secondGraph (G[2])

1: DataSet<GradoopId> graphId = secondGraph.getGraphHead()
2: .map(new Id<G>())
3: 
4: DataSet<V> newVertices = firstGraph.getVertices()
5: .filter(new NotInGraphBroadcast<V>())
6: .withBroadcastSet(graphId, GRAPH_ID)
7: 
8: DataSet<E> newEdges = firstGraph.getEdges()
9: .filter(new NotInGraphBroadcast<E>())
10: .withBroadcastSet(graphId, GRAPH_ID)
11: .join(newVertices)
12: .where(new SourceId<E>().equalTo(new Id<V>())
13: .with(new LeftSide<E, V>())
14: .join(newVertices)
15: .where(new TargetId<E>().equalTo(new Id<V>())
16: .with(new LeftSide<E, V>())

Exclusion

1: Person name: Alice
   born: 1984
2: Band name: Metallica
   founded: 1981
3: Person name: Bob
4: Person name: Eve
5: Band name: AC/DC
   founded: 1973

1|Community|interest:Heavy Metal
2|Community|interest:Hard Rock

1|Person
2|Band
3|Person
4|Person
5|Person

likes since: 2013
likes since: 2014
likes since: 2015
likes since: 2014
knows
IMPLEMENTATION OF GRAPH GROUPING

\[ \text{Map} \rightarrow \text{V} \rightarrow \text{GroupBy(1)} + \text{GroupReduce}\]
- Assign vertices to groups
- Compute aggregates
- Create super vertex tuples
- Forward updated group members

\[ \text{Filter} + \text{Map} \rightarrow \text{V'} \]
- Extract super vertex tuples
- Build super vertices

\[ \text{Filter} + \text{Map} \rightarrow \text{V_3} \]
- Extract group members
- Reduce memory footprint

\[ \text{Map} \rightarrow \text{E} \rightarrow \text{E_1} \rightarrow \text{Join}\]
- Replace Source/TargetId with corresponding super vertex id

\[ \text{GroupBy(1,2,3)} + \text{GC + GR* + Map} \rightarrow \text{E'} \]
- Assign edges to groups
- Compute aggregates
- Build super edges

*Requires worker communication
1. Extract **subgraph** containing only *Persons* and *knows* relations

2. **Transform** *Persons* to necessary information

3. Find communities using **Label Propagation**

4. **Aggregate** vertex count for each community

5. **Select** communities with more than 50K users

6. **Combine** large communities to a single graph

7. **Group** graph by *Persons* *location* and *gender*

8. **Aggregate** vertex and edge count of grouped graph
1. Extract **subgraph** containing only **Persons** and **knows** relations

2. Transform **Persons** to necessary information

3. Find communities using **Label Propagation**

4. Aggregate vertex count for each community

5. Select communities with more than 50K users

6. Combine large communities to a single graph

7. Group graph by Persons **location** and **gender**

8. Aggregate vertex and edge count of grouped graph

---

```java
return socialNetwork
// 1) extract subgraph
.subgraph((vertex) -> {
    return vertex.getLabel().toLowerCase().equals(person);
}, (edge) -> {
    return edge.getLabel().toLowerCase().equals(knows);
})
// project to necessary information
.transform((current, transformed) -> {
    transformed.setLabel(current.getLabel());
    transformed.setProperty(city, current.getPropertyValue(city));
    transformed.setProperty(gender, current.getPropertyValue(gender));
    transformed.setProperty(birth, current.getPropertyValue(birth));
    return transformed;
}, (current, transformed) -> {
    transformed.setLabel(current.getLabel());
    return transformed;
})
// 2a) compute communities
.callForGraph(new GellyLabelPropagation<GraphHeadPojo, VertexPojo, EdgePojo>(maxIterations, label))
// 2b) separate communities
.splitBy(label)
// 4) compute vertex count per community
.apply(new ApplyAggregation<>(vertexCount, new VertexCount<GraphHeadPojo, VertexPojo, EdgePojo>())
// 5) select graphs with more than minClusterSize vertices
.select(g) -> {
    return g.getPropertyValue(vertexCount).getLong() > threshold;
}
// 6) reduce filtered graphs to a single graph using combination
.reduce(new ReduceCombination<GraphHeadPojo, VertexPojo, EdgePojo>()
// 7) group that graph by vertex properties
.groupBy(Lists.newArrayList(city, gender))
// 8a) count vertices of grouped graph
.aggregate(vertexCount, new VertexCount<GraphHeadPojo, VertexPojo, EdgePojo>())
// 8b) count edges of grouped graph
.aggregate(edgeCount, new EdgeCount<GraphHeadPojo, VertexPojo, EdgePojo>())
```

---

https://git.io/vgozj
BENCHMARK RESULTS

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphalytics.1</td>
<td>61,613</td>
<td>2,026,082</td>
</tr>
<tr>
<td>Graphalytics.10</td>
<td>260,613</td>
<td>16,600,778</td>
</tr>
<tr>
<td>Graphalytics.100</td>
<td>1,695,613</td>
<td>147,437,275</td>
</tr>
<tr>
<td>Graphalytics.1000</td>
<td>12,775,613</td>
<td>1,363,747,260</td>
</tr>
<tr>
<td>Graphalytics.10000</td>
<td>90,025,613</td>
<td>10,872,109,028</td>
</tr>
</tbody>
</table>

- 16x Intel(R) Xeon(R) 2.50GHz (6 Cores)
- 16x 48 GB RAM
- 1 Gigabit Ethernet
- Hadoop 2.6.0
- Flink 1.0-SNAPSHOT
BENCHMARK RESULTS 2

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphalytics.1</td>
<td>61,613</td>
<td>2,026,082</td>
</tr>
<tr>
<td>Graphalytics.10</td>
<td>260,613</td>
<td>16,600,778</td>
</tr>
<tr>
<td>Graphalytics.100</td>
<td>1,695,613</td>
<td>147,437,275</td>
</tr>
<tr>
<td>Graphalytics.1000</td>
<td>12,775,613</td>
<td>1,363,747,260</td>
</tr>
<tr>
<td>Graphalytics.10000</td>
<td>90,025,613</td>
<td>10,872,109,028</td>
</tr>
</tbody>
</table>

- 16x Intel(R) Xeon(R) 2.50GHz (6 Cores)
- 16x 48 GB RAM
- 1 Gigabit Ethernet
- Hadoop 2.6.0
- Flink 1.0-SNAPSHOT
## COMPARISON

<table>
<thead>
<tr>
<th>Data Model</th>
<th>Graph Database Systems: Neo4j, OrientDB</th>
<th>Graph Processing Systems: (Pregel, Giraph)</th>
<th>Graph Dataflow Systems: (Flink Gelly, Spark GraphX)</th>
<th>Extended PGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus</td>
<td>queries</td>
<td>analytic</td>
<td>analytic</td>
<td>analytic</td>
</tr>
<tr>
<td>Query Language</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>(yes)</td>
</tr>
<tr>
<td>Graph Analytics</td>
<td>(no)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Scalability</td>
<td>vertical</td>
<td>horizontal</td>
<td>horizontal</td>
<td>horizontal</td>
</tr>
<tr>
<td>Workflows</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Persistency</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic Graphs / Versioning</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Data Integration</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>(yes)</td>
</tr>
<tr>
<td>Visualization</td>
<td>(yes)</td>
<td>no</td>
<td>no</td>
<td>limited</td>
</tr>
</tbody>
</table>
- Motivation
  - graph data
  - requirements

- Graph data systems
  - graph database systems
  - distributed graph processing systems (Pregel, etc.)
  - distributed graph dataflow systems (GraphX, Gelly)

- Gradoop
  - architecture
  - Extended Property Graph Model (EPGM)
  - implementation and performance evaluation

- Open challenges
CHALLENGES

- Graph data allocation and partitioning
- Benchmarking and evaluation of graph data systems
- Graph-based data integration and knowledge graphs
- Analysis of dynamic graphs
- Interactive graph analytics
distributed graph processing depends on suitable graph allocation/partitioning

- minimize communication for graph analysis
- load balancing

goal: balanced vertex distribution with minimal number of edges between partitions (edge cut)

- vertex cut: balanced edge distribution with minimal replication of vertices (PowerGraph, Spark GraphX)
hash-based vertex partitioning prevalent but not optimal
  - vertex neighbors frequently in different partitions -> high communication overhead

multilevel graph partitioning (e.g., METIS)
  - expensive to determine / static

newer approaches for adaptive allocation
  - Stanton/Kliot (KDD2012), Mondal/Deshpande (Sigmod2012), Huang/Abadi (VLDB2016)
many comparative evaluations between graph DBMS and graph processing systems (Han - VLDB14, Lu -VLDB14, ...)

- many differences in considered systems, workloads, configurations, etc
- early systems using Map/reduce or Giraph are outperformed by newer graph processing systems
- few results for Spark GraphX, Flink Gelly

Benchmark efforts for graph data analysis

- e.g., LinkBench, LDBC, gMark
- only few results so far
need to integrate diverse data from different sources (or from data lake) into semantically expressive graph representation

- for later graph analysis
- for representing background knowledge (knowledge graphs)

traditional tasks for data acquisition, data transformation, data cleaning, schema / entity matching, entity fusion, data enrichment / annotation

most previous work for RDF data, but not for property graphs
"Business Intelligence on Integrated Instance Graphs (BIIIG)" (PVLDB 2014)

(1) Graph transformation
(2) Graph integration
(3) Subgraph Isolation

Integrated Instance Graph

Data Sources

meta data

Domain expert

Business Transaction Graphs
INTEGRATION SCENARIO

source: Andreas Thor
- graphs like social networks, citation networks, road networks etc change over time
  - need to efficiently update/refresh analysis results (graph metrics, communities/clusters, ...)
  - streaming networks vs slowly evolving networks
  - fast stream analysis vs. analysis of series of graph snapshots
- many initial studies on specific aspects but no comprehensive system for analysis of dynamic graphs
need to support both interactive graph queries / exploration + graph mining

OLAP-like graph analysis functionality
- Multi-level, multidimensional grouping and aggregation
- need for extended (nested) graph model?

visual analytics for big graphs
- data reduction techniques for visualization (sampling, multi-level grouping, ...)

INTERACTIVE GRAPH ANALYTICS
AGENDA

- Motivation
  - graph data
  - requirements

- Graph data systems
  - graph database systems
  - distributed graph processing systems (Pregel, etc.)
  - distributed graph dataflow systems (GraphX, Gelly)

- Gradoop
  - architecture
  - Extended Property Graph Model (EPGM)
  - implementation and performance evaluation

- Open challenges
REFERENCES


Gradoop

- M. Junghanns, M. Kießling, A. Averbuch, A. Petermann, E. Rahm: Cypher-based Graph Pattern Matching in Gradoop. Proc. ACM SIGMOD workshop on Graph Data Management Experiences and Systems (GRADES), 2017
- A. Petermann; M. Junghanns: Scalable Business Intelligence with Graph Collections. it - Information Technology Special Issue: Big Data Analytics, 2016
- A. Petermann, M. Junghanns, R. Müller, E. Rahm: BIIIG : Enabling Business Intelligence with Integrated Instance Graphs. Proc. 5th Int. Workshop on Graph Data Management (GDM 2014)
- A. Petermann, M. Junghanns, R. Müller, E. Rahm: Graph-based Data Integration and Business Intelligence with BIIIG. Proc. VLDB Conf., 2014